Chapter 2

Finite-state syntactic analysis

In this chapter the finite-state automaton is introduced, and it is explained why finite-state techniques are suitable for use in natural language processing. Of particular importance is the distinction between competence and performance in human language processing. Against this background the idea of finite-state approximation is introduced, and several methods of approximation are described. This is followed by a discussion of the suitability of these methods for the current research and a proposal for an alternative method.

2.1 Finite-state automata

Finite-state automata (FSA) are used in a wide variety of fields to describe processes. They are used in predicting the weather, in controlling robots, in recognition of spoken language, in compiling computer programs, and so on. In all cases, the automaton also describes a language, in the sense that it dictates how single actions can be combined to form processes and thus how symbols can be combined to form acceptable sequences of symbols. The class of languages described by finite-state automata is known as the class of regular languages. In the complexity scale of languages known as the Chomsky hierarchy [28], represented in table 2.1, the regular languages are the least powerful class. That finite-state automata are nonetheless often used to model processes suggests that these processes are either of regular form, or close enough to regular form so that a finite-state description is an acceptable approximation.

In order to be better able to talk about finite-state automata and their role in natural language processing (NLP), the next section will define the concept of the finite-state automaton and the variations on it that are most relevant to the research described in following chapters. First an informal

Grammar	Languages
Type-0	Recursively enumerable
Type-1	Context-sensitive
Type-2	Context-free
Type-3	Regular

Table 2.1: The Chomsky hierarchy of languages.

sketch will be presented, which is followed by a more formal definition.

2.1.1 Informal definition

A finite-state automaton may be thought of as an input-reading machine that has a finite number of internal states. At any moment, the machine is in one of its states. The input to the machine consists of sequences of symbols from an alphabet associated with the machine. For every state a set of *state transitions* into other states is defined. Every transition is governed by a symbol from the alphabet, and a transition can only be used if the symbol that governs the transition is the current input symbol. That symbol is then "consumed" as the machine moves from the one state to the other, using the transition. In addition, a transition may be governed by the ϵ symbol that represents the empty string. Such an ϵ-transition may be used without consuming an input symbol. One state is assigned the role of initial state, and one state is assigned the role of final state. (Particular definitions allow for sets of initial and final states, however in this case the simpler definition is used as explained in section 2.1.2.) A series of symbols is recognized by the automaton when, starting from the initial state and reading the symbols in the input sequence one by one, a sequence of transitions can be used that brings the automaton into the final state, and there are no input symbols left to be read. The automaton is said to define a language, which is the language of all strings that can be recognized by the automaton.

Graphical representation of finite-state automata

A finite-state automaton can be defined in terms of its set of states, its alphabet and its transitions. In this manner the different subtypes of automata will be formally defined in section 2.1.2. A way of presenting an automaton is through the use of a graph. In this type of graph, nodes represent the states of the automaton and arcs between the nodes represent state transitions. Arcs are labeled with the symbols associated with the transitions. The ini-

tial state is marked with a small arrow, and the final state is marked with a double circle. States may be numbered or otherwise labeled for explanatory reasons.

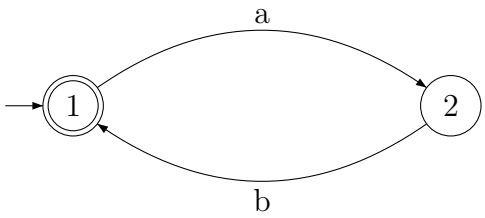

Figure 2.1: Finite-state automaton defining the language $(ab)^*$.

Figure 2.1 shows an automaton that defines the regular language $(ab)^*$. This is the language of all words that consist of zero or more occurrences of the string ab. A word consisting of no symbols will be accepted by the automaton since in this particular case the starting state is also the final state. Any number of repetitions of the string ab will be accepted through first taking the transition from state 1 to state 2, reading the a, followed by the transition from state 2 back to state 1, reading the b. If there is no input left at this point, the input string is accepted as the automaton is in the final state. If repetitions of ab are left in the input, recognition continues as described. If at any point during this process an input symbol is encountered for which there is no transition available, or all input has been read yet the automaton is not in its final state, the input is not accepted. Examples of input leading to these two different types of rejection are abb and aba respectively.

Deterministic versus non-deterministic

If input is fed to the automaton in figure 2.1, the characters in the input will determine exactly the actions of the automaton. For example, when the automaton is in state 1, and an a is read, it will move to state 2. There is only one transition from state 1 that is labeled with an a. If in general an automaton has for every state exactly one transition for each input symbol, it is called a *deterministic* automaton. The alternative is a *non-deterministic* automaton in which the input characters do not necessarily determine what states the automaton will go through. In such an automaton, a single state may feature several transitions associated with the same input symbol.

Figure 2.2 shows an example of a non-deterministic automaton. It defines the language $(ab^+)^*$, which is the language consisting of repetitions of the

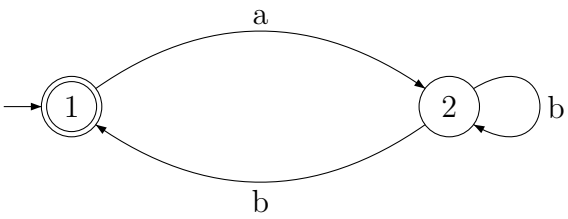

Figure 2.2: Non-deterministic finite-state automaton defining the language $(ab^+)^*$.

pattern in which an a is followed by one or more b's. Examples of words in this language are ab, abb, and $abbbbabbb$. The non-determinism in the automaton compared to the automaton in figure 2.1 is caused by the additional transition from state 2 to itself, labeled with a b. If the automaton is in state 2, and the current input symbol is a b, both a transition to state 1 as well as a transition to state 2 are possible. In this situation the input alone does not uniquely determine the next state.

Apart from two transitions from the same state sharing the same input symbol, non-determinism can also be caused by ϵ-transitions. If a state features an ϵ-transition, the automaton can decide at this point in processing to use the ϵ-transition instead of a standard transition that may also be possible. In doing so, the internal state is changed, but no input symbol is consumed. Deterministic automata are required not to contain ϵ-transitions.

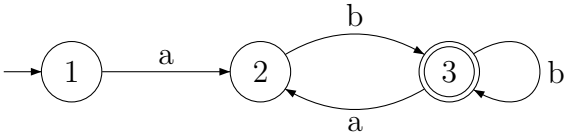

Figure 2.3: Deterministic finite-state automaton defining the language $(ab^+)^*$.

Writing down a non-deterministic automaton is typically easier than finding its deterministic counterpart. As an example, a deterministic version of the automaton for the language $(ab^+)^*$ is given in figure 2.3. In an actual implementation a deterministic automaton is preferable. Using a deterministic finite-state automaton, the time needed for recognizing a given input is linear

in the length of the input; the size of the automaton itself is not important. With a non-deterministic automaton, the recognition time is related to the size of the automaton. However, there exists an algorithm to create an equivalent deterministic automaton out of a non-deterministic automaton ([57], p. 69).

Weighted versus non-weighted

A second distinction that is important and also relevant to this work is that between weighted and non-weighted automata. In the above description of the finite-state automaton, weights did not play a role; the non-weighted automaton either accepts or rejects a sequence of symbols. In a weighted automaton, transitions are associated with weights as well as with symbols. This kind of automaton assigns a weight to input in addition to merely accepting or rejecting it, by combining weights that correspond to subsequent transitions; such an automaton therefore defines a weighted language. If the requirement is met that for every state the weights associated with the transitions from that state sum up to one, the automaton defines a probabilistic language; in this case the probabilities of all strings accepted by the automaton, as computed by multiplication over transition probabilities, also sum up to one.

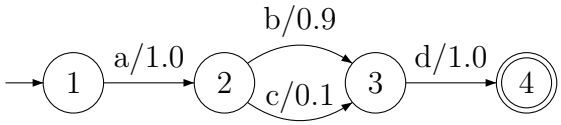

Figure 2.4: Weighted finite-state automaton defining the language $abd \cup acd$.

Figure 2.4 represents a weighted automaton, defining the language that contains just the two strings *abd* and *acd*. In this particular example the transition weights can be viewed as probabilities, therefore the automaton defines a probabilistic language. The transition probabilities are indicated to the right of the corresponding transition symbols. It can be seen that the transition from state 2 to 3 that is labeled with a *b* has a probability of 0.9 assigned to it, which is to be compared with the probability of 0.1 that is assigned to the alternative transition that is labeled with a *c*. The probabilities of strings are computed by multiplying the transition probabilities as

a sequence of states in the automaton is visited. This means that according to this automaton the probability of *abd* is 0.9 ($1.0 \times 0.9 \times 1.0$) whereas the probability of *acd* is 0.1 ($1.0 \times 0.1 \times 1.0$).

2.1.2 Formal definition

Typically one will find in the literature definitions of the deterministic automaton featuring multiple final states and definitions of the non-deterministic automaton featuring, in addition, multiple initial states. In order to simplify the definition and the presentation and use of the automata, the different kinds of automata are defined here to have exactly one initial state and one final state. These definitions are equivalent to the more extensive definitions, in that an automaton with multiple initial states can be changed into an equivalent automaton with just one initial state by first adding the new initial state, and then adding ϵ-transitions from the new initial state to each of the former initial states. In a similar manner a set of former final states can be linked to a single final state. (In practice both modifications to the model can be accomplished by adding respectively `beginning-of-sentence` and `end-of-sentence` markers to the data on which the models are based: if these are unique symbols that mark the beginning and end of every sentence, the model based on this data will automatically feature a single initial and a single final state. This approach avoids the use of ϵ-transitions.)

Deterministic finite-state automaton

The non-weighted deterministic finite-state automaton (DFSA) is a 5-tuple $M = (Q, \Sigma, \delta, q_1, q_f)$ where Q is the finite set of states, Σ is the finite input symbol alphabet, δ is the transition function $Q \times \Sigma \rightarrow Q$, q_1 is the initial state such that $q_1 \in Q$, and q_f is the final state such that $q_f \in Q$. In order to define the recognition of a sequence by M, we define δ^* as the extended transition function $Q \times \Sigma^* \rightarrow Q$. A string W is recognized by M if and only if $\delta^*(q_1, W) = q_f$. All strings thus recognized by M form the language defined by M. This is by definition a regular language.

Non-deterministic finite-state automaton

The non-deterministic finite-state automaton (NFSA) is defined as a 5-tuple $M = (Q, \Sigma, \Delta, q_1, q_f)$. This is the same 5-tuple as in the case of the DFSA, with the exception that the transition function δ is now a transition relation Δ that is defined as $Q \times (\Sigma \cup \epsilon) \times Q$. Next, Δ^* is defined as a subset of

$Q \times (\Sigma \cup \epsilon)^* \times Q$, the reflexive and transitive closure of Δ. A string W is recognized by M if and only if $(q_1, W, q_f) \in \Delta^*$.

Weighted deterministic finite-state automaton

The weighted deterministic finite-state automaton (WDFSA) is defined as a 5-tuple $M = (Q, \Sigma, \delta, q_1, q_f)$, identical to the DFSA except that the transition function δ is defined as $Q \times \Sigma \times \mathbb{R} \rightarrow Q$ so that every transition is associated with a weight. Recognition of a sequence by M is defined as in the case of the DFSA, and in addition a weight is associated with a sequence W recognized by M that is defined as the product of the weights on the transitions used during recognition of W.

Weighted non-deterministic finite-state automaton

The weighted non-deterministic finite-state automaton (WNFSA) is defined as a 5-tuple $M = (Q, \Sigma, \Delta, q_1, q_f)$ that is similar to the WDFSA except for the transition function δ that is replaced by a transition relation Δ defined as $Q \times (\Sigma \cup \epsilon) \times \mathbb{R} \times Q$. Recognition of a sequence W by M proceeds as in the case of the NFSA. In addition, a weight is assigned to the recognized sequence that is the sum of the separate weights for all paths through the automaton that correspond to W, representing all possible ways in which M can recognize W.

2.2 Motivation for the finite-state paradigm

In this section arguments will be provided in favor of using finite-state automata in NLP, and for syntactic analysis in particular. First, it will be explained what makes finite-state techniques attractive in general, leading to their application in many different fields. Second, their application in syntactic analysis will be discussed with specific arguments and examples.

2.2.1 General motivation for finite-state techniques

In a deterministic FSA, every input symbol that is read will prompt the automaton to use the appropriate transition from the current state into the next state, if this transition is available. When all n symbols of an input string have been processed, the automaton will have used at most n transitions and the input is either accepted or rejected based on whether the current state is a member of the set of final states. This linear complexity renders finite-state techniques efficient for use in implemented computational systems.

For every non-deterministic automaton, there exists a deterministic version. In addition, an automaton can also be minimized. This means we can define for every automaton the minimal (most compact) deterministic automaton that still defines the same language.

Furthermore, an important aspect of the class of regular languages is that it is closed under the operations of union, intersection, Kleene star, concatenation and complementation. This means that a set of finite-state automata, each representing a particular regular language, may be combined under these operations into a larger automaton that still represents a regular language. This allows for the creation of separate modules, in the form of automata, that are combined into a more versatile automaton that still has the attractive properties of a finite automaton.

Consequently finite-state approaches are used successfully in different areas within the larger field of NLP. These areas include phonology, morphology and syntax. An extensive overview of finite-state techniques in NLP is [76]. With respect to syntax, a popular application involving finite-state automata is that of part-of-speech tagging, which plays an important role in this work. Section 2.2.2 will consider the use of finite-state methods in syntax.

2.2.2 Finite-state techniques in syntactic analysis

In judging the use of finite-state techniques in the processing of natural language on the level of its syntax, we will not ignore constructions found in natural languages that are not representable through finite-state means. Granting the existence of these constructions, we will argue that finite-state techniques are nevertheless appropriate and useful in syntactic analysis.

Grammars underlying natural languages typically allow for constructions that are not regular. Many languages allow for the type of construction called *center-embedding* (or *self-embedding*), and Dutch, as well as other languages, additionally features the *cross-serial dependency* construction. A case of center-embedding is given in example (1). Pairs of noun phrases and verb phrases that are syntactically dependent of each other are recursively divided through the embedding of phrases of a similar structure. The result is a palindrome sequence with a structure of the form $w_1w_2w_2w_1$, or in general WW^R, where W^R is the reverse of sequence W. Example (2) is a case of cross-serial dependencies, where noun phrases and their corresponding verbs are again separated. The resulting sequence of the form $w_1w_2w_3w_1w_2w_3$ is an example of an utterance in a *copy language*, which is generally represented as WW.

(1) de kat die_1 de muis die_2 de kaas at_2 ving_1
the cat that the mouse that the cheese ate caught
the cat that caught the mouse that ate the cheese

(2) de bewaker die_1 de man_2 de gevangene_3 liet_1 helpen_2
the guard that the man the prisoner let help
ontsnappen_3
escape
the guard that let the man help the prisoner to escape

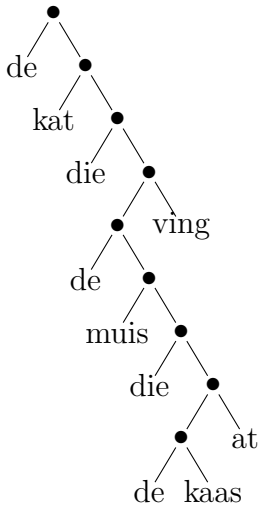

Figure 2.5: A parse tree corresponding to example (1).

A simplified parse tree for example (1) is given in figure 2.5. The palindrome and the copy language utterance share the characteristic that elements in the left half of the sequence are systematically related to elements in the right half. It can be seen in figure 2.5 how the parse tree for the center embedding example contains a large tree fragment for the phrase *de muis die de kaas at* that separates the noun *kat* from its verb *ving*. In order to properly analyze the combination of the noun and the verb, the noun must remain in memory until the whole of the embedded tree fragment has been analyzed, at which point the verb is finally addressed. This type of dependency cannot be expressed through finite-state means for sequences of unrestricted length, since the automaton does not have a mechanism for storing information about an unbounded number of elements that were seen in the past.

This will now be proven by contradiction. In this proof, the language $a^n b^n$ for $n >= 0$, in which words consist of any number of a's followed by the same number of b's, is used as an example.

Proof. Assume a finite-state automaton can be created that defines the language $a^n b^n$ for $n >= 0$. The number of states in the automaton is i. By its definition, the automaton must accept $a^k b^k$ for $k = i$. Each symbol read means a transition into a next state, thus after having accepted k symbols, starting from the initial state, $k + 1$ states will have been visited (namely the initial state and one state for every symbol). Since $k = i$, one state must have been visited (at least) twice. Assume this is state q. State q was visited in accepting symbols at two distinguished positions i_1 and i_2 with $i_1, i_2 \leq k$. State q thus represents both sequences of a's of length i_1 as well as of length i_2. Now assume the automaton is in state q and starts reading b's. From that state the automaton has to accept sequences of b's of length i_1 as well as of length i_2. Therefore, one of the total sequences accepted by the automaton is $a^{i1} b^{i2}$ for $i_1 \neq i_2$, which is not in the language $a^n b^n$. Thus this automaton is not a correct definition of $a^n b^n$ for $n >= 0$. □

The palindrome structure can be described using a *context-free* grammar, a type of grammar that is one level higher than the regular languages in the Chomsky hierarchy. In the case of the copy language, not even a context-free description is possible; here, a grammar of *context-sensitive* power is required.

In practice however, language produced by humans has characteristics that are reminiscent of finite-state processes. The distinction between what is possible in theory and what is produced in practice is the distinction between language *competence* and language *performance* as made by Chomsky [29]. Chomsky suggested that the competence grammar might be approximated by a finite-state device, proposing that a theory of grammars should provide, in addition to a set of possible grammars $G_1, G_2, \ldots$, a function $g(i, n)$ that returns the description of a finite-state automaton with memory capacity n that accepts sentences and produces structural analyses assigned to these sentences in accordance with competence grammar G_i.

Center-embedding and cross-serial dependencies are clearly part of the competence grammar of languages that allow such constructions. Although these constructions are non-regular, there are three aspects of human language processing that suggest that human language performance is finite-state in nature:

1. Humans have a limited amount of working memory available, as do finite-state automata.

2. Humans have problems processing cases of center-embedding and cross-serial dependencies, which are non-regular constructions.

3. Human language processing efficiency is reminiscent of finite-state efficiency.

The amount of working memory available to any organism or machine is physically limited. The number of dependencies between elements in a sentence has to be finite in order for such an entity to properly analyze or produce it. This implies that actual cases of non-regular syntactic constructions, occurring in utterances by humans, can always be represented by some finite-state automaton. To argument 1 one could object that although memory available to humans is indeed not unlimited, it is still of considerable size, rendering the fact that it is finite less interesting when considering the length of sentences in human language; one could speculate that relatively to the length of the average sentence, the amount of available memory could be treated as infinitely large.

However, in this respect argument 2 about center-embedding is stronger than the first argument, and it suggests that the above speculation is incorrect. It can be observed that the admissible number of embeddings of this kind in a sentence that is still easily understandable is quite small. Example (1) is easy to understand, but the meaning of example (3) is only retrieved after studying the sentence for a longer period than would typically be required for a sentence of this length, and through actively combining the actors with their actions.

(3) De man$_1$ die$_2$ de hond die$_3$ de kat die$_4$ de muis die$_5$ de
The man that the dog that the cat that the mouse that the
kaas at$_5$ ving$_4$ beet$_3$ riep$_2$ stierf$_1$.
cheese ate caught bit called died
The man that called the dog that bit the cat that caught the mouse that ate the cheese died.

Pulman [71] defines an automaton as *strictly finite-state* when its number of states is not only finite but also small, in contrast to the trivial definition where only the finiteness is considered. Example (3) makes it clear that, although the amount of memory available to humans is considerable, the center-embedding construction quickly leads to problems when the number of embeddings is increased beyond even a small count. This suggests that human performance on this point is finite-state, not just in the trivial sense as implied by argument 1, but also in the strict sense.

The third point mentioned here to support the idea that human language performance is finite-state is concerned with processing efficiency. It can be observed that humans understand longer sentences just as easily as shorter ones. In both cases, understanding seems to be almost instantaneous, needing about as much time as is needed to utter the sentence in the first place. Finite-state automata show a linear increase in the number of steps necessary for recognizing longer sequences of symbols. More powerful types of grammar are less efficient: practical algorithms used to analyze sentences with context-free grammars in the worst case require a number of steps that is proportional to the length of the sentence to the power of three. (The algorithm by Valiant [89] has a slightly lower worst case complexity but is not practical due to the fact that it always operates at the worst case complexity.) This difference in efficiency between the class of context-free grammars and human language processing, and the apparent similarity on this point between human language processing and finite-state processing, suggests that the finite-state formalism is a more likely model of human language performance than more powerful formalisms.

2.3 Methods of finite-state approximation

It can be concluded from section 2.2 that if finite-state methods are to be used in syntactic analysis of natural language, the result can only be a model of language performance, which was argued in the previous section to be finite-state, and an approximation of language competence, which is taken to be of at least context-free power.

What separates the class of regular languages from the class of context-free languages, is the ability in the case of the context-free grammars to remember an unbounded number of symbols that featured in an earlier part of an expression to be accepted or generated, and to use the information stored in processing subsequent parts of the same expression. As illustrated in figure 2.7, this ability corresponds to the operation of self-embedding. Self-embedding is a combination of embedding and recursion. Recursion in itself is not a sufficient condition for a grammar to be of greater than finite-state power, and neither is embedding. As an example, the grammar in figure 2.6 features embedding of the term X in S, as well as a recursive definition of X itself. It defines the language ac^*b, which is regular.
In this language the number of a's and b's to the left and right of the embedded structure is fixed. This relation can therefore be represented in a finite-state automaton. Contrary to this, through self-embedding a similar correspondence can be defined for unbounded numbers of a's and b's, as in

$$\begin{array}{lcl} \text{S} & \rightarrow & \text{a X b} \\ \text{X} & \rightarrow & \text{c X} \\ \text{X} & \rightarrow & \epsilon \end{array}$$

Figure 2.6: Grammar generating a regular language with embedding.

the grammar in figure 2.7 that defines the language $a^n b^n$ which was shown to be non-regular in section 2.2.2.

$$\begin{array}{lcl} \text{S} & \rightarrow & \text{a S b} \\ \text{S} & \rightarrow & \epsilon \end{array}$$

Figure 2.7: Grammar generating a non-regular language.

Regular grammars or finite-state automata are thus able to remember a bounded number of past events, while context-free grammars, or in general grammars of greater than regular power, may represent structures that require remembering an unbounded number of past events. In this context, the technique referred to as approximation of context-free grammars with finite-state grammars is concerned with reproducing as much as possible the infinite memory of non-regular grammars in a finite-state automaton. This idea can be expressed through Chomsky's function $g(i, n)$ that was already mentioned in section 2.2.2. For a competence grammar G_i and a given amount of memory n, this function returns the description of a finite-state approximation of G_i that uses memory proportional to n. The parameter n can be seen as indicating how close the approximation is to the original: although n will always be finite, and the resulting automaton will always have a finite memory, larger values for n will result in an automaton that better approximates a device with unbounded memory.

As an example the language $a^m b^m$ is considered. A regular grammar may only recognize this language correctly for $0 >= m >= j$ for some fixed j. Allowing the use of a smaller or larger amount of memory, j may be respectively lower or higher. As the amount of available memory nears infinity, j will near infinity, and the language defined by the automaton will be an increasingly accurate approximation of $a^m b^m$ for unbounded m.

Approximation may result in an automaton that defines a subset of the original language, or a superset, or neither. In the subset case, the accuracy of the approximation increases as an increasingly large subset of the target language is represented. For the superset approximation, accuracy increases

as a smaller superset is defined. If the approximation represents neither a subset nor a superset of the target language, evaluation of the approximation has to be done along an empirical route by taking precision and recall into account, which are measurements of the approximation based on respectively the incorrect constructions and the missing constructions with respect to the target language.

It has to be noted here that there are many approaches to finite-state syntax in which finite-state automata are created, sometimes manually, without reference to a more powerful grammar. Examples are cascades of finite-state transducers [2, 16, 46]. (Transducers are automata that translate input to output.) A single transducer may also be used repeatedly on its own output [75]. These methods are not regarded as methods of approximation.

What follows is an overview of different approaches to finite-state approximation of context-free grammars. An overview of this research field given by Nederhof [64] was used as a guideline.

2.3.1 Approximation through RTNs

Nederhof [65] proposed an approach to finite-state approximation of context-free grammars inspired by recursive transition networks (RTNs). RTNs [102] can be seen as defining context-free languages. Nederhof describes how a RTN can be constructed for a given CFG. A finite-state automaton is constructed for every nonterminal and its defining rules in the CFG. The resulting set of automata form the RTN. An example CFG, $\mathcal{G}$, is given in figure 2.8. The corresponding RTN, $\mathcal{R}$, which in this case consists of just one automaton for the single nonterminal S, is given in figure 2.9. It can be seen that the automaton features a recursive call to itself (that is, to the automaton defining S) on the transition from state q_2 to q_3.

$$\begin{array}{lcl} \mathrm{S} & \rightarrow & \mathrm{a}\ \ \mathrm{S}\ \ \mathrm{b} \\ \mathrm{S} & \rightarrow & \epsilon \end{array}$$

Figure 2.8: Example CFG $\mathcal{G}$.

The approximation is constructed by combining the automata of the RTN into one automaton and approximating any occurrences of recursion using ϵ-transitions. If states q_A and q'_A are respectively the start and end state of an automaton recognizing nonterminal A, a transition (q, A, q') is replaced by two transitions (q, ϵ, q_A) and (q'_A, ϵ, q'). Figure 2.10 shows this operation being applied to $\mathcal{R}$. The end result, after removal of ϵ-transitions, is the FSA

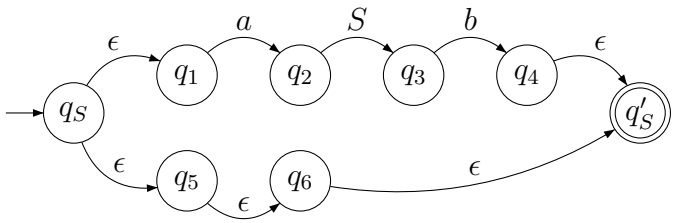

Figure 2.9: RTN $\mathcal{R}$ corresponding to CFG $\mathcal{G}$.

$\mathcal{F}$ in figure 2.11. $\mathcal{F}$ describes the language a^*b^*, a superset of the original non-regular language a^nb^n.

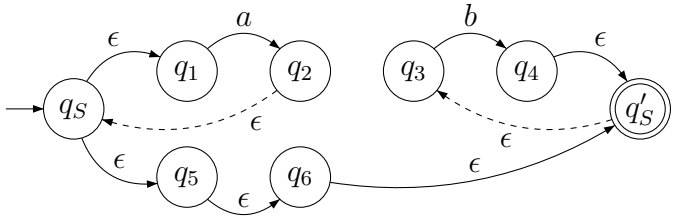

Figure 2.10: FSA approximating recursion in RTN $\mathcal{R}$.

Nederhof describes a parameterized variant of this approach. It is noted that states correspond to *items*, where items are *dotted rules* that represent steps in processing. A dotted rule of the form $A \rightarrow \alpha \bullet \gamma$ corresponds to a rule $A \rightarrow \alpha\gamma$ from the original grammar to which a dot in the right hand side of the rule has been added indicating how far the derivation of A has proceeded.

In the parameterized variant, each state is associated not only with an item I, but with a history H of items as well. Every item in H is of the form $A \rightarrow \alpha \bullet B\gamma$. For all states that are used in the sub-automaton describing nonterminal B, this history will be identical. If there are several different histories possible for reaching B, there will be equally many copies of the sub-automaton defining B, associated with their respective histories. This way, limited depths of recursion are implemented by "folding out" the original automaton. The length of the history H is limited according to parameter d, such that $|H| < d$. Further details of the construction are not repeated here, but an example is given in figure 2.12.

In this example, $d = 2$. The sub-automaton at the top of figure 2.12 is

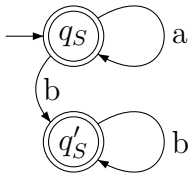

Figure 2.11: FSA $\mathcal{F}$, the automaton of figure 2.10 after minimization and determinization.

associated with history $H = [\epsilon]$. The bottom sub-automaton is associated with history $H = [S \rightarrow a \bullet Sb]$, defining a nonterminal S that is recursively embedded in another S.

The language described by this new automaton is a slightly more accurate approximation of $a^n b^n$ than the previous approximation $a^* b^*$, in that if there are zero a's, there will also be zero b's, and vice versa. As d is increased, the result will be an automaton that describes the recursive structure of the language $a^n b^n$ exactly for larger n, making for an increasingly accurate approximation.

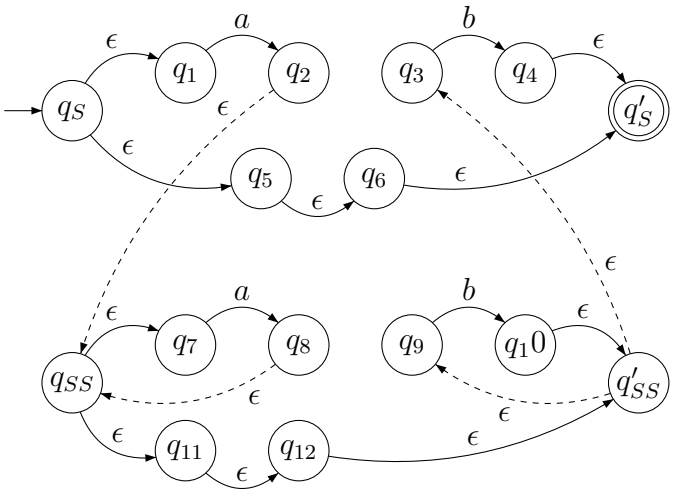

Figure 2.12: FSA that is a more accurate approximation of the RTN $\mathcal{R}$ resulting from applying the parameterized algorithm ($d = 2$) to $\mathcal{G}$.

2.3.2 Approximation through grammar transformation

Nederhof [64] describes an alternative approach that modifies the rules of a CFG in such a way that the grammar is restricted to describing a regular language, by ruling out self-embedding.

The steps of the transformation are not repeated here in their literal form. Instead, the rules and their effect are described by use of the concept of *spines* in a parse tree, as is done in [64]. A spine of a parse tree is defined as a path from the root of the tree to a leaf of the tree. Figure 2.13 shows a parse tree in which circles are drawn around nodes that make up a spine.

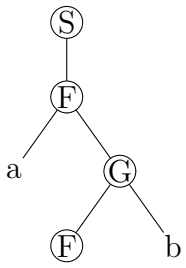

Figure 2.13: Example parse tree with spine indicated by circles.

All possible spines for a given CFG are considered, and traversed from top to bottom. On coming across a nonterminal A in the spine, it is tagged with a set of pairs (B, C). B is a nonterminal occurring in the spine above A. Thus we are considering a production $B \rightarrow A$. C is a set that indicates whether elements have occurred to the left or right of nonterminal B, in which case C contains respectively an l and an r.

If $C = \{l, r\}$, we are considering a production of the form $B \rightarrow \alpha A \beta$ for non-empty α and β. The effect of the grammar transformation rules is that in the transformed grammar, a node A labeled with a set C that contains $(A, \{l, r\})$ is not allowed to be part of the tree, as this would constitute a case of self-embedding: A has occurred once already in the parse tree, and elements were since then produced to the left and to the right of this occurrence.

In the tree in figure 2.13, the F at the bottom of the spine would be labeled with the set $\{(S, \{l, r\}), (F, \{l, r\}), (G, \{r\})\}$. The derivation from G to F would not be possible in the transformed grammar due to $(F, \{l, r\})$ being part of the labeling for the lower node F.

The resulting automaton is a subset approximation of the original language. It is noted that this method may produce huge grammars, as the set of labeled nonterminals grows very large.

An alternative parameterized method with lower complexity is also presented. A parameter d indicates a maximum number of times that elements are added to both the left and right of the spine. In between these events, cases of left recursion followed by right recursion and right recursion followed by left recursion are allowed. Nonterminals are tagged with a set containing either l, r, or both, but without referring to a previous nonterminal. If after having seen d rules that add something to the left and right there are yet more rules of this type occurring further down along the spine, these are not added to the transformed grammar. This method has lower complexity than the previously described approach, but it is also less precise in that it may cancel derivations that are not cases of self-embedding.

2.3.3 Approximation through restricted stack

Storage and retrieval of past events can be implemented by means of a *stack*. Elements can be put on top of the stack one by one, and retrieved in reversed order by taking them from the top of the stack. An automaton equipped with an unbounded stack (or *push-down store*) can recognize the non-regular language $a^n b^n$. Such an automaton is called a *push-down automaton* (PDA).

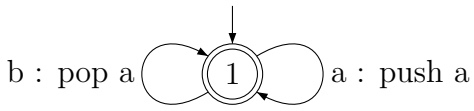

Figure 2.14: Push-down automaton defining the language $a^n b^n$.

Figure 2.14 shows a push-down automaton that recognizes the language $a^n b^n$. The transitions are labeled with input symbols and stack operations. The two operations allowed are *push* and *pop*, which will respectively place a symbol on top of the stack, and remove a symbol from the top of the stack. An input sequence is accepted if the automaton is in a final state, there is no input left, and the stack is empty. Table 2.2 shows the current state, the remaining input, stack contents, and subsequent actions taken by the PDA in figure 2.14 in recognizing the input sequence *aaabbb*.

Methods of approximation can use the push-down automaton as a basis. The general idea is to limit the size of the stack, effectively turning the push-down automaton into a finite-state automaton. Two approaches using this

state	input	stack	action
1	aaabbb	{ }	read a, push a
1	aabbb	{a}	read a, push a
1	abbb	{a,a}	read a, push a
1	bbb	{a,a,a}	read b, pop a
1	bb	{a,a}	read b, pop a
1	b	{a}	read b, pop a
1		{ }	accept

Table 2.2: Current state, remaining input, stack contents, and actions of the PDA in figure 2.14 during recognition of *aaabbb*.

idea are presented, differing with respect to whether a subset or a superset of the target language is defined by the resulting approximation.

Subset approximation

Pulman [71] notes that in aiming for a psychologically plausible model for human language performance, left and right recursive structures are problematic. Although these are typically considered regular, their internal structure is non-regular. This can be illustrated with example (4). The brackets that indicate structure are related to each other in a non-regular (center-embedded) way.

(4) $[_1$the cat that caught $[_2$the mouse that ate $[_3$the cheese$]_3]_2]_1$

Pulman presents a parsing procedure which is described as similar to the shift-reduce algorithm [5]. It differs from this procedure in that it can also handle incomplete constituents, the effect of which is that it can be seen as using the left-corner parsing strategy. (The left-corner strategy defines the order in which the parse tree is constructed as first recognizing the left-most element of a rule's right hand side as in bottom-up parsing, on the basis of which the remainder of the rule's right hand side is predicted as in top-down parsing; these remaining elements are then parsed using the same alternation of bottom-up recognition and top-down prediction.)

With the above observation concerning recursion in mind, as a first step towards an approximation it is stated that no unlimited recursion of any kind is allowed. A limit L is implemented in such a way that if the parser calls a recursive routine more than L times, the information related to the previous L calls is lost. However, humans do not show problems in actually dealing with left and right recursive (or left and right embedding) structures. Pulman's

parser accommodates this. Left embedding does not pose a problem for the parser under the restriction as described above, as in this case the use of the stack never increases beyond a constant amount. Right embedding can pose a problem to the parser, as stack usage will in some cases grow proportional to the height of the parse tree. The parser is therefore extended with the *clear* operation. This operation is applied to the parser's stack and combines a VP needing an S with an S (and an S needing a VP with a VP) into just one stack entry if the two entries are adjacent, and only when the stack size is beyond some predefined value. This operation serves to keep right recursive structures under the recursion limit L.

The finite-stateness of the parser is thus ensured through maintaining the limit L on the amount of recursion allowed, restricting the application of center-embedding while leaving left embedding and right embedding unrestricted. Pulman's approximation will accept a subset of the original language. Center-embedded expressions will be exactly recognized for a fixed maximal recursive depth, while such expressions containing recursion beyond the maximal depth will be rejected.

The asymmetry in efficiency of processing of left-branching and right-branching structures using the left-corner parsing strategy is also noted by Resnik [74]. A parser uses memory to store nodes that are as yet incomplete; either their parent node or a child node of the node at hand has not been created. Resnik gives parse tree examples of left-branching, center-embedded, and right-branching structures. The left-branching construction, using the left-corner strategy, requires a constant amount of memory. The right-branching construction, using the same strategy, requires an amount of memory that is proportional to the height of the parse tree.

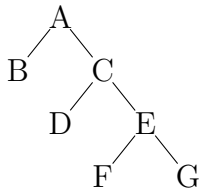

Figure 2.15: A right-branching parse tree.

Resnik then proposes a left-corner parser using *arc-eager enumeration*. In the case of *arc-standard enumeration*, the arc between a node and its child node will only be created once the child's subtree has been parsed.

Figure 2.15 shows a right-branching tree. The arc between nodes A and C will only be created once the subtree dominated by E has been parsed. However, using arc-eager enumeration, the arc between A and C will be created as soon as C is predicted (which is immediately after creation of node D). This renders node A complete even though E still has to be parsed. It is shown that this strategy, formalized as a push-down automaton, requires memory proportional to the height of the parse tree only for cases of center embedding. The use of arc-eager enumeration is equivalent to Pulman's use of the *clear* operation; both are cases of *tail-recursion optimization*.

A similar idea is presented by Johnson [49]. Johnson proposes a left-corner grammar transformation. This transforms a context-free grammar into a grammar that enforces the left-corner parsing strategy, including tail-recursion optimization to ensure that right-recursive structures can be analyzed with a finite memory. The method is implemented as a finite-state transducer that for a given input sentence produces a sequence of productions representing a parse of the input with respect to the transformed grammar. Following this, the transformation is inverted, resulting in a parse tree corresponding to the previous end result but written down in terms of the original grammar.

Superset approximation

Pereira and Wright [68] note that grammars powerful enough for accurate syntactic analysis are typically not efficient enough for an application such as speech recognition. They propose to use separate grammars for recognition and interpretation, the grammar for recognition to be implemented as a finite-state grammar. The recognition grammar should not cancel readings allowed by the interpretation grammar that is applied after it, and should, at the same time, enforce as many of the constraints in the interpretation grammar as it can. It is noted that in the process of constructing both these grammars at the same time it can be difficult to ensure these requirements are met. Therefore Pereira and Wright choose to create the recognition grammar automatically as a finite-state approximation of the context-free interpretation grammar.

$$\begin{array}{lcl} \mathrm{S} & \rightarrow & \mathrm{a}\ \ \mathrm{S}\ \ \mathrm{b} \\ \mathrm{S} & \rightarrow & \epsilon \end{array}$$

Figure 2.16: Example CFG $\mathcal{G}$.

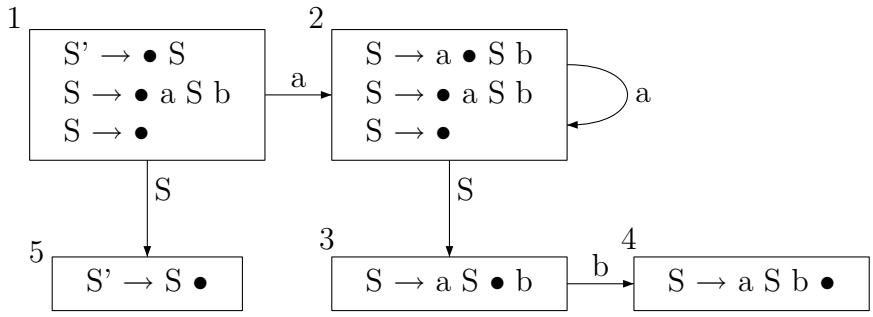

Figure 2.17: Characteristic finite-state machine $\mathcal{M}$ for CFG $\mathcal{G}$.

Their method creates a finite-state acceptor based on the *characteristic finite-state machine* (CFSM) of a context-free grammar. The CFSM is the finite-state control for a shift-reduce recognizer, or LR(0) parser. As an example, the CFSM for the CFG in figure 2.8, repeated here in 2.16, is given in figure 2.17. The states of the CFSM contain sets of items, or dotted rules. Transitions between states represent applications of the *shift* operation, and items of the form $A \rightarrow \gamma\bullet$ allow for application of the *reduce* operation. The construction of the CFSM, which is omitted here, is described by Pereira and Wright and in [4, p. 221] and [8], for example.

state	input	stack	action
0	aabb	$\{\}$	shift a
1	abb	$\{\langle 1, a\rangle\}$	shift a
1	bb	$\{\langle 1, a\rangle, \langle 2, a\rangle\}$	reduce S → •
2	bb	$\{\langle 1, a\rangle, \langle 2, a\rangle, \langle 2, S\rangle\}$	shift b
3	b	$\{\langle 1, a\rangle, \langle 2, a\rangle, \langle 2, S\rangle, \langle 3, b\rangle\}$	reduce S → a S b •
2	b	$\{\langle 1, a\rangle, \langle 2, S\rangle\}$	shift b
3		$\{\langle 1, a\rangle, \langle 2, S\rangle, \langle 3, b\rangle\}$	reduce S → a S b •
4		$\{\langle 1, S\rangle\}$	accept

Table 2.3: Steps taken in recognizing *aabb* using $\mathcal{M}$.

Table 2.3 shows the steps taken in recognizing the string *aabb* using the CFSM in figure 2.17. The table shows the current state, remaining input, stack contents, and the performed action which leads to the situation described by the next line in the table. The stack contains pairs of the form $\langle s, a\rangle$ where s is a state and a is a symbol. A transition from state q_1 to q_2

associated with symbol a corresponds to a shift operation and adds the pair $\langle q_1, a\rangle$ to the stack. The reduction operation for a rule of the form $A \rightarrow \gamma\bullet$ removes pairs corresponding to symbols γ from the top of the stack and takes the transition labeled with A from the state s associated with the last pair popped from the stack (or from the current state if no elements are popped, as in the case of reduction $S \rightarrow \bullet$) and pushes the pair $\langle s, A\rangle$.

Pereira and Wright propose to turn the shift-reduce recognizer into an FSA by removing the stack and replacing reduction moves with ϵ-transitions. This technique is called *flattening*. Normally a reduction leads to another state based on the elements that were popped from the stack, as explained above. Now this is achieved through an ϵ-transition directly to that state. The result of flattening the CFSM in figure 2.17, followed by removal of transitions labeled with nonterminals and after minimization and determinization, is the FSA in figure 2.18 that describes the language $a^+b^+ \cup \{\epsilon\}$.

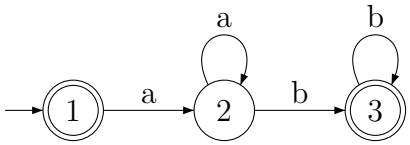

Figure 2.18: FSA resulting from flattening CFSM $\mathcal{M}$.

As a finite-state approximation of the non-regular language a^nb^n, this is acceptable. However, Pereira and Wright point out that even simple regular languages are often incorrectly reproduced using this method. They provide an example of a language consisting of the strings *aca* and *bcb*. Flattening of the corresponding CFSM leads to an FSA that recognizes not only these strings, but also *acb* and *bca*. The problem lies in the replacing of reductions with ϵ-transitions. The state where a reduction is applied may have been reachable from a number of different states, which were part of distinct paths that are now merged.

The suggested solution is to apply an operation of *unfolding* to the CFSM before the automaton is flattened, which means that each state is replaced by a set of states corresponding to possible stack configurations at that state. In this manner, the different ways in which the original state could have been reached are explicitly represented. The set of possible stacks is typically infinite due to loops in the CFSM, therefore an equivalence relation is used according to which different stacks are treated as one and the same stack, creating a finite number of stack classes. Two stacks are considered equivalent if they can be made similar by removal of loops, where loops are repeated

stack segments.

Rood [77] uses the same idea and adds the ability to specify the number of loops that is still considered in differentiating stacks. Here, states are labeled with the paths that led to them, where paths consist of alternating state numbers and input symbols. An unfolding criterion N is decided on, and paths containing a number of loops greater than N are assigned to one and the same infinite class of paths, represented by the path that contains $N + 1$ loops.

In general the resulting automaton will recognize a superset of the original language. Using Rood's method, the automaton is able to exactly recognize non-regular languages up to a recursive depth specified by the unfolding criterion. The algorithm described by Pereira and Wright [68] has been used in limited-domain speech recognition tasks.

2.3.4 Approximation using n-gram models

In *n-gram models* of language, the probability of the occurrence of a word is modeled as only depending on the identity of the previous $n - 1$ words. Models of this kind will be discussed in detail in chapter 3.

Stolcke and Segal [83] describe a method in which n-gram probabilities are computed directly from a stochastic context-free grammar (SCFG). This method is said to avoid problems related to learning n-gram models from annotated data, which include the estimation of probabilities for a large number of parameters, the requirement of a very large training corpus, the inability to directly model linguistic knowledge, and the relative difficulty of adding new information to a model.

The method estimates the probabilities for n-grams by computing their *expected frequencies* in sentences generated by the SCFG. The different ways in which a nonterminal (in particular the start symbol S) can produce a given substring are considered, and the respective probabilities are summed.

The implementation described by Stolcke and Segal computes bigram probabilities in time $O(N^2)$ at most, where N is the number of nonterminals in the grammar. The system is used in a speech-recognizer in the BeRP spoken-language system [52]. An experiment using a prototype system of BeRP resulted in a word recognition error rate of 35.1% when using bigram probabilities that were estimated from the training corpus, compared to 33.5% when using a combination of corpus-based probabilities and probabilities estimated from the SCFG as described above, in a proportion of 2500 : 200,000. It is concluded that in this experiment, in which a training corpus of 2500 sentences was used with an average length of 4.8 words, the method was of help in improving bigram probability estimates that were based on a

relatively small amount of data.

2.4 Discussion of methods of approximation

As described in chapter 1, this research aims at a finite-state approximation of the Alpino wide-coverage parser for Dutch [15]. In the previous section, several methods of finite-state approximation of a grammar of greater than finite-state power were described. With the intended source system in mind, the described methods are considered and an alternative and more suitable approach will be suggested, to be discussed in the next chapter.

2.4.1 Computational complexity

High complexity may be a problem in constructing finite-state automata that approximate more powerful grammars. Nederhof [64] found that in using the method of Pereira and Wright [68] to approximate a relatively small grammar of 50 rules, an amount of memory was required that rendered the process of determinization and minimization of the automata practically impossible. In this respect, the RTN methods were shown to be preferable. The automaton created using both the standard and parameterized RTN approach grows only moderately as the size of the source grammar increases.

2.4.2 Systems beyond context-free power

The techniques described are methods that approximate a context-free grammar. It was mentioned in section 2.2.2 that the cross-serial dependency type of construction cannot be represented with a context-free grammar. A system that tries to describe as precisely as possible a natural language that features cross-serial dependencies is likely to contain special machinery for this task. Thus in a practical setting where a system is designed to cover as much of a given language as possible, an approximation technique expecting a standard context-free grammar is not useful. Typically, this would require that one somehow compile the grammar into an intermediate context-free grammar, as in [68]. The left-corner transformation described by Johnson [49] is described as being easily adaptable to finite-state approximation of unification-based grammars, not requiring an intermediate context-free grammar, but this is not explained in detail.

2.5 Discussion

One of the goals of this work is to find a way of approximating an existing wide-coverage parser accurately in a finite-state model. The parser we have in mind uses a stochastic attribute-value grammar. In such systems in general, features are used that cannot be described in a context-free way. As described above, an intermediate context-free approximation would have to be used in order to be able to use a method of approximation that expects a context-free grammar as input.

In addition, systems used in a practical setting will encounter problems in parsing real text. For instance, various rule-based and/or stochastic heuristics may have to be implemented to cope with unknown words. These techniques will typically be separate modules, in principal unrelated to the grammar used in the main system. An approximation directly based on the grammar would not contain the heuristics. An example concerning the system to be approximated in this work, the Alpino parser, is the stochastic disambiguation model which is used to decide which parse for a given input is most likely the correct one. In approximating the system as a whole, we would like components such as these to be accounted for.

In the next chapter, the method known as *inference* will be presented. Inference consists of deriving a grammar from a collection of utterances. The process of inference will be used in chapter 4 to acquire an approximation of the source system. Instead of using an algorithm to directly reduce a grammar to finite-state power, the output of the system that uses the grammar is collected and used to construct a finite-state model. The approximation process is shown in figure 2.19.

In this manner, in principle nothing needs to be known about the nature of the original grammar and the parsing system that uses it. Techniques such as the stochastic disambiguation used to select the most probable analysis will be reflected in the system's output so that through inference, this component can be part of the approximation. This approach closely follows the idea of creating a finite-state model of language performance, as an approximation of language competence. The complete parsing system represents the competence, and a collection of sentences, originally produced by humans and now analyzed and annotated by the parser, represents linguistic performance.

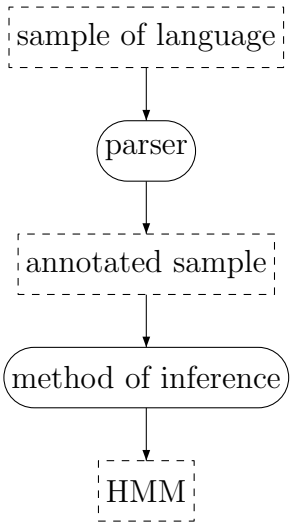

Figure 2.19: Approximation through inference.